Formal Verification of Floating-Point Programs

Sylvie Boldo
INRIA Futurs
Sylvie.Boldo@inria.fr

Jean-Christophe Filliâtre
CNRS
Jean-Christophe.Filliatre@lri.fr

ProVal, Parc Orsay Université, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract

This paper introduces a methodology to perform formal verification of floating-point C programs. It extends an existing tool for the verification of C programs, Caduceus, with new annotations specific to floating-point arithmetic. The Caduceus first-order logic model for C programs is extended accordingly. Then verification conditions expressing the correctness of the programs are obtained in the usual way and can be discharged interactively with the Coq proof assistant, using an existing Coq formalization of floating-point arithmetic. This methodology is already implemented and has been successfully applied to several short floating-point programs, which are presented in this paper.

1 Introduction

Critical applications, such as numeric simulations in nuclear physics or aeronautics, do require a high level of guarantee. But, as soon as floating-point computations are involved, the level of guarantee is usually rather low. This may be due to the difficulty of a sharp analysis of floating-point computations (rounding and exceptions). Even if each step is required to be correct by the IEEE-754 standard [23, 24], whole numeric programs' behaviors are often difficult to foresee, and exceptional behaviors disturb intuition.

Some algorithms have a pen and paper proof, but this has proved not to be enough [16, 20]. Some programs need a high level of reliability that only formal proofs can provide. Formal methods have been successfully used both for hardware-level and high-level floating-point arithmetic [2, 6, 21, 10, 11, 19, 7, 14].

This paper introduces a new approach to the formal verification of floating-point C programs, which consists in a specification language at the C source level and in a methodology to get verification conditions to be discharged with the Coq proof assistant [1]. This work is based on two existing works: the Caduceus tool for the verification of C programs [9, 15] and a Coq model of floating-point arithmetic [7]. Our main contribution is the extension of the Caduceus specification language to handle floating-point arithmetic. To our knowledge, such a specification language has never been designed before. The main difficulty lies in the fact that we want to express not only properties about what is actually computed (floating-point numbers) but also about what would be ideally computed (with real numbers and without any rounding) and even sometimes about what the user intends to compute (which may be only approximated by the program).

This paper is organized as follows. Section 2 gives an overview of the Caduceus tool. Then Section 3 introduces our extension to support floating-point arithmetic. Finally, Section 4 illustrates our verification methodology on several examples.

2 The Caduceus tool

Caduceus [9, 15] is a tool for the formal verification of C programs at the source code level. The properties that may be checked are of two kinds: first, the program may be checked to be free of threats (null pointer dereferencing or out-of-bounds array access) and second, it may be proved to satisfy functional properties given as annotations in the source. These annotations include functions's preconditions and postconditions, global invariants, loop invariants, etc., as usual in Hoare logic frameworks [12]. They are inserted in the source as comments of a specific shape `/*@...*/` and written in a first-order specification language largely inspired by the Java Modeling Language [17]. A significant part of ANSI C is supported, including pointer arithmetic and possible pointer aliasing.

Here is for instance a possible specification for a function `math_mod` computing the modulo:

```
/*@ requires
  @   x >= 0 && y > 0
  @ ensures
  @   0 <= \result < y &&
  @   \exists int d; x == d * y + \result
  @*/
int math_mod(int x, int y) { ... }
```

The keyword `requires` introduces the function's precondition, that is a property to be satisfied whenever the function is called. The keyword `ensures` introduces the function's postcondition, that is a property which must be established whenever the function returns. As one can notice, the specification language augments the usual C syntax (`&&`, ==, etc.) with new constructs such as `\result` denoting the value returned by the function, the existential quantifier `\exists`, etc. Contrary to JML, program annotations are not intended to be executable.

Once a C source code is adequately annotated, the Caduceus tool produces verification conditions. These are logical statements expressing that code is free of threats and fulfills the given specification. The verification conditions are then discharged using a general purpose theorem prover. A key feature of the Caduceus tool is its independence with respect to the prover. It indeed supports a wide panel of existing provers, ranging from interactive proof assistants such as Coq, PVS or Isabelle to automatic theorem provers such as Simplify, Yices or CVC Lite.

Before presenting our extension of the Caduceus tool to handle floating-point arithmetic, we need to give a few details about its architecture. Caduceus is actually built on top of another tool, called Why. The combination of these two tools is illustrated in Figure 1. Caduceus first translates the C source code into an intermediate code written in the syntax of the Why language. This in an alias-free Hoare logic-like language suitable for program verification. This intermediate program uses a first-order logic model of C programs (pointer arithmetic, layout of the memory heap, etc.) which is itself written in the syntax of the Why tool. This model is made of declarations of function symbols, predicates and axioms. Then the Why tool computes verification conditions and translates them to the native syntax of the various provers. Depending on the prover, the model can be simply axiomatized or fully implemented.

To add floating-point arithmetic support to Caduceus, we are going to operate at several levels:

- the specification language is extended with new annotations specific to floating-point arithmetic;
- the Why model is extended accordingly;
- the Why model is realized within the Coq proof assistant to allow the user to discharge the verification conditions.

The next section describes this extension.

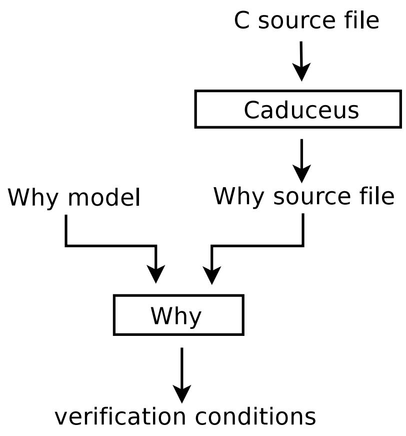

Figure 1. The Caduceus / Why combination

3 Adding floating-point arithmetic

3.1 A model of floating-point arithmetic

The main idea is the following: each floating-point value is modelled as a triple consisting of

- the floating-point number of the given type (typically single or double), as computed by the program;
- a real number, which would be the value if all previous computations were performed on real numbers and thus exact;
- and another real number, which is the value which should be *ideally* computed.

If x is a floating-point value, we refer to its three components respectively as the *floating-point* part, written x_f, the *exact* part, written x_e and the *model* part, written x_m.

For example, let us assume that we have a C variable `x` of type `double` and the following definition of the variable `y`

```
double y=1+x*(1+x/2);
```

Then the floating-point part of y is $\circ(1 + \circ(x_f \times \circ(1 + \circ(x_f/2))))$ where $\circ$ denotes the current rounding mode (by default, rounding to nearest, ties to even). The exact part of y is $1 + x_e + \frac{{x_e}^2}{2}$.

Contrary to the floating-point part and the exact part, the model part can be assigned an arbitrary real number. Thus it must be seen as a *ghost variable*[1] automatically assigned to each floating-point expression. In this example, it could be

[1]A ghost variable is a program variable which does not interfere with the program computations but is only used in the specification.

assigned the value $\exp(x_m)$ to express the intent to compute (an approximation of) the exponential of `x`. This example is developed in more details in Section 4.3.

When conditional instructions are involved, the test must be performed on the floating-point part, in accordance to the program execution. Thus the exact and model parts are computed following the program path defined by the floating-point part. This is mandatory, as programs such as example 4.2 would not even have a resulting value if the tests were performed on the exact part.

Rounding modes are correctly handled: if the rounding mode is not modified by the program, it has a known value, which can be specified on Caduceus's command line and which defaults to rounding to nearest, ties to even. If it is modified by the program, then it becomes a dynamic value and floating-point expressions are interpreted according to its current value.

3.2 Syntax of annotations

Our abstract model of floating-point arithmetic being defined, we now extend the Caduceus specification language to give access to this model. We only describe here the new constructs introduced to deal with floating-point arithmetic; a comprehensive description of the Caduceus specification language can be found in its reference manual [15].

First, all operations appearing inside annotations are interpreted as exact operations on real numbers. Thus 0.1 is exactly $\frac{1}{10}$ and $1 + 0.1$ is a real addition which value is thus 1.1. Similarly, one can use π, $\sqrt{2}$ or $\exp(1)$ in annotations with their usual mathematical meaning.

A floating-point program variable is interpreted as its floating-point part and implicitly converted to a real number. Two constructs are introduced to access the other two parts of the model: `\exact`(e) is the exact part of the floating-point expression e and `\model`(e) its model part.

In addition, several derived constructs are introduced to ease the writing of annotations:

- `\abs(`e`)` or `|`e`|`: the absolute value of the real expression e;
- `\round_error(`e`)`: a shortcut for $|e -$ `\exact`$(e)|$, that is the difference between the floating-point part of e and its exact part;
- `\total_error(`e`)`: a shortcut for $|e -$ `\model`$(e)|$, that is the difference between the floating-point part of e and its model part;
- e_1`^^`e_2: the exponentiation of real numbers;
- `\sqrt(`e`)`: the square root of the real expression e.

3.3 Why specification of the model

We now specify the model of floating-point arithmetic as a Why theory, that is a set of types, constants and function declarations. It must be seen as the *interface* of our model. In order to use a given prover to discharge the verification conditions, this model has to be realized first (see next section). An excerpt of this theory is given in Appendix (the full theory can be found in the Caduceus distribution in file `lib/why/floats.why`).

The main elements of this theory are the following:

- a type for the rounding modes and five constants of this type (the four rounding modes of IEEE-754, plus the rounding to nearest, ties away from zero of the revision of the IEEE-754 standard [13]);
- the single and double types;
- the basic operations (addition, subtraction, division, square root, negation, absolute value, etc.) on these types, used in the interpretation of the program computations;
- the functions to access the floating-point part, the exact part and the model part, used in the interpretation of the annotations;
- the rounding functions (to interpret the constants);
- the maximum float allowed in each type, as a real number;
- the coercions between the different types (single to double and double to single).

In addition, this Why theory also contains variants of the floating-point operations with overflow checks. Such operations perform the same computations as the ones above, but have preconditions enforcing the user to prove that the resulting value is not greater than the maximal float allowed in the given type. Thus we can actually choose between two different models: one model where we *assume* that there is no overflow during the program execution, and one model where we have to *prove* that there is no overflow (a Caduceus command-line option switches between the two models). We could also implement a third model where overflows are taken into account (and thus where `NaN` is an explicit value). But this would require much work for the very few situations where a program is correct in spite of overflows. We do not consider floating-point flags and exceptions at all. It could be added to this model but it would make it much more complex, as each operation should also return flags. Since such flags are rarely and hardly accessed in C programs, we prefer not handling them.

Note that we do handle underflows: our model of floating-point numbers contains subnormal numbers and tiny enough values will be rounded into subnormals. This means that we may have to add hypotheses on some programs such as "the input values are normal floats".

3.4 Coq implementation of the model

At this point, we are now able to produce the verification conditions for any prover supported by the Why tool. But none of these provers is able to discharge any of the verification conditions, since our model is merely a set of declarations without definitions. Thus the last step of the verification chain consists in implementing the floating-point model for a particular prover, in order to be able to conduct proofs. We focus here on the Coq proof assistant [1].

Our Coq implementation of the model is based on an existing formalization of the IEEE-754 standard by M. Daumas, L. Rideau and L. Théry [7]. This formalization is radix-independent and format-independent. To correspond to floating-point processors, we choose the radix 2 and we use single and double floating-point numbers that have the required mantissa and exponent size. To be in accordance with the Why model, the Coq implementation of a floating-point number is a three fields record composed of:

- the floating-point value, of type `float` as defined in [7];
- a proof that the float fits in the given format (single or double precision);
- an exact value of type `R`, where `R` is the Coq type of real numbers;
- a model value of type `R`.

For the rounded operations, we define only the basic operations defined by the IEEE-754 standard (addition, subtraction, multiplication, division and square root), plus opposite and absolute value. These are defined using rounding functions that were previously developed [3]. We also define functions for jumping from one format to another (without rounding for single to double, but with rounding from double to single).

We also need a function to round a constant value: if 0.1 appears in a program, it will be seen as the single (or double) floating-point number with a floating-point part that is the rounded (with the current rounding mode) of the real value $\frac{1}{10}$ and an exact and a model parts that are exactly $\frac{1}{10}$.

It is important to notice that our architecture based on a model written in the syntax of the Why tool offers a great flexibility. It is indeed very easy to switch to another floating-point computational model, such as radix 10 or fixed-point arithmetic. All the user has to do is to replace the Coq implementation of the Why model. Similarly, it is easy to switch to another prover, as soon as a floating-point formalization is available (e.g. HOL and PVS [6]).

4 Examples

This section illustrates the use of our verification framework on several short floating-point programs.

4.1 Sterbenz theorem

The well-known Sterbenz theorem [22] states that if x and y are floating-point numbers not too far away ($y/2 \leq x \leq 2\,y$), then $x - y$ is computed exactly.

This can be formally stated as the specification of a single precision C function computing $x - y$:

```
/*@ requires y/2 <= x <= 2*y
  @ ensures  \result == x-y
  @*/
float Sterbenz(float x,float y){
  return x-y;
}
```

Let us denote $x = (x_f, x_e, x_m)$ and $y = (y_f, y_e, y_m)$. The result of the function call, denoted `\result`, is the triple $(\circ(x_f - y_f), x_e - y_e, x_m - y_m)$. The annotation states that the floating-point part is exactly the subtraction of the floating-point parts of x and y, that is $\circ(x_f - y_f) = x_f - y_f$.

Since there is no modification of the rounding mode in the source code, all computations are assumed to be in rounding to nearest, ties to even and the rounding is statically set in the verification conditions. A Caduceus command-line option can be used to specify a different rounding mode. When run on the above source code, Caduceus produces a single verification condition, which is the following statement:

$$\begin{array}{l} \forall x : \texttt{single}.\ \forall y : \texttt{single}. \\ \texttt{s\_to\_r}(y)/2 \leq \texttt{s\_to\_r}(x) \leq 2 \times \texttt{s\_to\_r}(y) \Rightarrow \\ \quad \texttt{s\_to\_r}(\texttt{sub\_single}(\texttt{nearest\_even}, x, y)) = \\ \quad \texttt{s\_to\_r}(x) - \texttt{s\_to\_r}(y) \end{array}$$

where all inequalities and equalities are on real numbers, `s_to_r` is the projection to access the floating-point part (as a real number), `sub_single` the subtraction on the type `single` and `nearest_even` the rounding mode (see the Appendix for the signatures of all these symbols). This verification condition is easily discharged with a dozen lines of Coq tactics using previous results from [7].

This function could be given a different, strengthened specification. Indeed, if we know beforehand that x and y have no rounding error, i.e. are exact representations of real numbers, then so is the result. This can be formally expressed as follows:

```
/*@ requires y/2 <= x <= 2*y
  @   && \round_error(x)==0
  @   && \round_error(y)==0
  @ ensures
  @   \round_error(\result)==0
  @*/
double Sterbenz(double x,double y){
  return x-y;
}
```

This variant was also proved correct using a few lines of Coq tactics.

4.2 Malcolm's algorithm

In the early seventies, Malcolm and Gentleman proposed an algorithm to deduce from some given computations the characteristics of the floating-point system (radix, precision, rounding mode, etc.) [18]. The most famous of these algorithms is the following one, which computes the radix of the floating-point system:

```
1. $A \leftarrow 2$
2. while $A \neq A + 1$ do $A \leftarrow A \times 2$
3. $B \leftarrow 1$
4. while $((A + B) - A) \neq B$ do $B \leftarrow B + 1$
5. return $B$
```

In our case the result should be 2, since we are considering IEEE-754 double precision. Nevertheless, the simple fact that this algorithm does terminate is a challenge in itself. Indeed, the first loop multiplies a variable A by 2 until A equals $A + 1$. This would clearly never halt if the computations were error-free.

To annotate the C implementation of the algorithm above, we need to introduce first two functions over real numbers:

```
/*@ logic int IRNDD(real s) */
/*@ logic int my_log(real s) */
```

Such `logic` declarations make Caduceus aware of the existence of such logical functions. They are defined on the prover side and do not require any definition at the source code level. These are two Coq functions from $\mathbb{R}$ to $\mathbb{Z}$ defined as $\texttt{IRNDD}(x) = \lfloor x \rfloor$ and $\texttt{my\_log}(x) = \lfloor \frac{\ln(x)}{\ln(2)} \rfloor$.

Then we can annotate a C version of Malcolm's algorithm:

```
/*@ ensures \result == 2 */
double malcolm() {
  double A, B;
  A=2;
  /*@ assert A==2 */

  /*@ invariant A == 2 ^^ my_log(A)
    @    && 1 <= my_log(A) <= 53
    @ variant (53-my_log(A)) */
  while (A != (A+1)) A*=2;

  /*@ assert A == 2 ^^ (53) */

  B=1;
  /*@ assert B==1 */

  /*@ invariant B == IRNDD(B)
    @    && 1 <= B <= 2
    @ variant (2-IRNDD(B)) */
  while ((A+B)-A != B) B++;

  return B;
}
```

Let us detail the annotations:

- The postcondition states that the result is the expected radix, that is 2.
- Two assertions state that A is 2 (resp. B is 1) after the assignment `A=2` (resp. `B=1`). Though it may seem obvious and redundant, it actually states that the *exact* parts of A and B are 2 and 1 respectively, i.e. that A and B are computed exactly.
- The first loop is annotated with an invariant stating that A is exactly of power of 2 and that its logarithm is between 1 and 53. It is also given a variant to guarantee its termination. The variant is here a nonnegative decreasing integer, namely $53 - \texttt{my\_log}(A)$.
- Right after the first loop, we can prove that $A = 2^{53}$, which is also inserted as an annotation.
- The annotation of the second loop is much simpler since we know that this loop will be executed exactly twice, B taking the values 1 and 2 successively. The only difficulty is here to state in the invariant that B is always an *integer* that is either 1 or 2.

When run on this annotated source code, Caduceus produces four verification conditions. All of them were proved correct using 163 lines of Coq tactics.

It is important to notice that this is not a proof that the Malcolm's algorithm returns the radix of the floating-point system. We knew beforehand that the radix was 2 and we used this information in the annotations to make the proof simpler.

4.3 An exponential

The purpose of this last example is to illustrate the use of the third component of our floating-point abstractions, which we called the *model part*.

It is a toy example computing the exponential of a double-precision float x around zero. A small Taylor polynomial is evaluated using Horner's rule, namely $1 + x(1 + x/2)$. In this case, the computation will be faithful [4, 5] so the rounding and total error can be bounded. More precisely, the source code can be annotated as follows:

```
/*@ requires |x| <= 2 ^^ (-3)
  @ ensures
  @ \model(\result)==exp(\model(x))
  @  && (\round_error(x)==0
  @        => \round_error(\result)
  @              <= 2 ^^ (-52))
  @  && \total_error(\result)
  @        <= \total_error(x)
  @              + 2 ^^ (-51)
*/
double my_exp(double x) {
  double y=1+x*(1+x/2);
  /*@ \set_model y exp(\model(x)) */
  return y;
}
```

Let us detail the annotations:

- The precondition states that the input must be small enough: the function will not compute an approximation of the exponential for $x = 100$. We require $|x| \leq 2^{-3}$.
- The postcondition then states that
  - the output is a float whose model part (its ideal value) is the exponential of the (ideal value of the) input;
  - the rounding error inside the function is bounded by 2^{-52};
  - the total error of the function (i.e. the difference between the floating-point value and the ideal value) is bounded by 2^{-51} plus the total error of the input.

It may seem somewhat redundant to set the model part to $\exp x$ in the code and to state the corresponding equality in the postcondition. As far as the correctness of `my_exp` is concerned, this is indeed a tautology, but it will be needed when calling `my_exp`.

On the annotated code above, Caduceus produces a single verification condition, which reads as follows:

$$\begin{array}{l}
\forall x : \texttt{double}.\ |\texttt{d\_to\_r}(x)| \leq 2^{-3} \Rightarrow \\
\forall y : \texttt{double}.\ \forall y_1 : \texttt{double}. \\
y = \texttt{add\_double}(\texttt{nearest\_even}, \\
\quad \texttt{r\_to\_d}(\texttt{nearest\_even}, 1), \\
\quad \texttt{mul\_double}(\texttt{nearest\_even}, x, \\
\qquad \texttt{add\_double}(\texttt{nearest\_even}, \\
\qquad \texttt{r\_to\_d}(\texttt{nearest\_even}, 1), \\
\qquad \texttt{div\_double}(\texttt{nearest\_even}, x, \\
\qquad \texttt{r\_to\_d}(\texttt{nearest\_even}, 2))))) \Rightarrow \\
y_1 = \texttt{double\_set\_model}(y, \exp(\texttt{d\_to\_model}(x))) \Rightarrow \\
\texttt{d\_to\_model}(y_1) = \exp(\texttt{d\_to\_model}(x)) \wedge \\
(\texttt{double\_round\_error}(x) = 0 \Rightarrow \\
\quad \texttt{double\_round\_error}(y_1) \leq 2^{-52}) \wedge \\
\texttt{double\_total\_error}(y_1) \leq \\
\quad \texttt{double\_total\_error}(x) + 2^{-51}
\end{array}$$

This has not yet been proved in Coq. Indeed, we do not want to perform an interactive proof but we rather plan to generate a proof automatically using the tool Gappa [8]; this is discussed in the next section.

Note that we choose on this example an annotation style where the function *requires* that $|x| \leq 2^{-3}$, which means that a call `my_exp(1)` would generate the unprovable verification condition $1 \leq 2^{-3}$. To be able to call this function on big inputs (but with no guarantee on the output) we could adopt a more liberal specification with no precondition and a postcondition of the shape $|x| \leq 2^{-3} \Rightarrow \ldots$. This is similar to the usual choice of defensive programming versus programming with assumptions.

5 Conclusions and perspectives

We have presented a formal verification framework for floating-point programs. From C programs annotated at the source code level, we get verification conditions that can be manually discharged using the Coq proof assistant. This work extends the existing tool for the verification of C programs Caduceus [15] with a new set of annotations to express properties of floating-point programs. On the prover side, it is based on a existing formalization of the IEEE-754 standard [7]. This verification methodology is already implemented and has been successfully used on several programs (which are quite short but require complex proofs).

Our main contribution is a specification language for floating-point programs. Such a specification is indeed a decisive key to the security of numerical programs: even if the annotations are not formally proved, the mere fact of precisely specifying what should do each function, what is its error bound, etc., is a huge step towards program correctness. Additionally, we provide a way to perform the full formal proof using the Coq proof assistant, and an open framework which is amenable to other floating-point models and/or other provers.

We are aware that the set of examples presented here is not large enough to convince that our set of annotations is adapted to all needs. Nevertheless, our annotations have been devised to be applicable to as many cases as possible. They will be used in the CerPAN project (`http://www-lipn.univ-paris13.fr/CerPAN/`) whose aim is to prove the soundness of programs from numerical analysis and especially a program computing the derivatives of a multivariable function.

A limitation of our current work is that it can only be applied to programs using basic operations (addition, subtraction, multiplication, division and square root). Many numerical programs heavily rely on elementary functions such as exponential or sine/cosine. These functions are harder to formalize than division as they are not standardized. Indeed, one may get different results depending on the processor used thus one has to turn the processor's reference manual into formal statements (e.g. "the maximal error on the exponential is $0.50\cdots01$ ulp"). Even if such formalizations are theoretically feasible, this would require a lot of work to be repeated for each platform.

Another limitation is the assumption that each double precision computation is performed on exactly 64 bits, which is not always the case on recent Intel architectures, where it can be computed using 80 bits. This feature would be very hard to formalize, as which computations are performed on 64 or 80 bits can not be predicted.

Another perspective is the use of the Gappa tool [8] which automatically generates formal proofs of floating-point properties using interval arithmetic. Gappa cannot solve all the verification conditions but could be of great help in formally proving error bounds or the absence of overflows. For instance, Gappa would automatically discharge the verification conditions of example 4.3. Moreover, Gappa builds Coq proofs that can be checked automatically, making the combination of interactive and automatic proofs trustworthy.

Acknowledgements. The authors wish to thank Marc Daumas and Guillaume Melquiond for constructive discussions about the possible type of annotations. The authors are also grateful to Micaela Mayero and François Clément for careful readings of this paper.

References

[1] The Coq Proof Assistant. `http://coq.inria.fr/`.

[2] G. Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, 1989.

[3] S. Boldo. Bridging the gap between formal specification and bit-level floating-point arithmetic. In C. Frougny, V. Brattka, and N. Mller, editors, *Proceedings of the 6th Conference on Real Numbers and Computers*, pages 22–36, Schloss Dagstuhl, Germany, 2004.

[4] S. Boldo and M. Daumas. A simple test qualifying the accuracy of horner's rule for polynomials. *Numerical Algorithms*, 37(1-4):45–60, 2004.

[5] S. Boldo and C. Muñoz. Provably faithful evaluation of polynomials. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing*, volume 2, pages 1328–1332, Dijon, France, Apr. 2006.

[6] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *1995 International Workshop on Higher Order Logic Theorem Proving and its Applications*, Aspen Grove, Utah, 1995. supplemental proceedings.

[7] M. Daumas, L. Rideau, and L. Théry. A generic library of floating-point numbers and its application to exact computing. In *14th International Conference on Theorem Proving in Higher Order Logics*, pages 169–184, Edinburgh, Scotland, 2001.

[8] F. de Dinechin, C. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 1318–1322, 2006.

[9] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, Nov. 2004. Springer-Verlag.

[10] J. Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[11] J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design*, pages 217–233, Austin, Texas, 2000.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[13] IEEE standard for floating-point arithmetic. Floating-Point Working Group of the Microprocessor Standards Subcommittee of the Standards Committee of the IEEE Computer Society, 2004. Work in progress.

[14] C. Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, Computer Science Department, Saarland University, Saarbrucken, Germany, 2002.

[15] Jean-Christophe Filliâtre, Claude Marché and Thierry Hubert. The Caduceus tool for the verification of C programs. `http://caduceus.lri.fr/`.

[16] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, 1985.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[18] M. A. Malcolm. Algorithms to reveal properties of floating-point arithmetic. *Commun. ACM*, 15(11):949–951, 1972.

[19] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating point hardware. *Intel Technology Journal*, 3(1), 1999.

[20] J. Rushby and F. von Henke. Formal verification of algorithms for critical systems. In *Proceedings of the Conference on Software for Critical Systems*, pages 1–15, New Orleans, Louisiana, 1991.

[21] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[22] P. H. Sterbenz. *Floating point computation*. Prentice Hall, 1974.

[23] D. Stevenson et al. A proposed standard for binary floating point arithmetic. *IEEE Computer*, 14(3):51–62, 1981.

[24] D. Stevenson et al. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, 1987.

Appendix: Why Theory

We give here a small excerpt of the Why theory corresponding to the floating-point model presented in this paper. The keyword `type` introduces a new datatype, which is abstract and purely applicative (`real` is the predefined type of real numbers). The keyword `logic` introduces a new function symbol with its arity. Finally, `parameter` declares a new program function (with no implementation). The syntax is $x_1 : \tau_1 \rightarrow \cdots \rightarrow x_n : \tau_n \rightarrow \{P\}\tau\{Q\}$ where the x_is are the function parameters, P the precondition, τ the type of the result and Q the postcondition.

```
(* rounding modes *)
type mode
logic nearest_even, to_zero, up, down,
      nearest_away : mode

(* the type of single precision floats *)
type single

(* operations *)
logic add_single :
  mode,single,single -> single
logic sub_single :
  mode,single,single -> single
...

(* coercions *)
logic s_to_r, s_to_exact, s_to_model :
  single -> real
logic r_to_s : mode,real -> single

(* overflow checks *)

logic max_single : real

parameter add_single_ :
  m:mode -> x:single -> y:single ->
  { abs_real(s_to_r(add_single(m,x,y)))
    <= max_single }
  single
  { result = add_single(m,x,y) }
...
```